**Numpy-II**

**Content**

* **Broadcasting**
* **Generating Random Numbers**
  + Uniformly Random Distribution
  + Random Normal Distribution
* **Shallow vs Deep Copy**
* **Aggregate Functions/ Universal Functions (ufuncs)**
  + sum()
  + mean(), median()
  + any(), all()
* **3D arrays**
* **Use Case: Image Manipulation using Numpy**
  + Opening an Image
  + Details of an image
  + Visualizing RGB Channels
  + Negative of an Image
  + Rotating an Image (Transposing a Numpy Array)
  + Trim Image
  + Saving ndarray as Image

import numpy as np

**Broadcasting**

**Do you remember the error thrown when we were trying to multiply a Matrix and a Vector using \* operator?**

A = np.arange(12).reshape(3, 4) # A is a 3x4 Matrix

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

v = np.array([1, 2, 3])

v

array([1, 2, 3])

**Doing A \* v:**

A \* v

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-72-42a27ee68a2a> in <module>

----> 1 A \* v

ValueError: operands could not be broadcast together with shapes (3,4) (3,)

**How about v \* A?**

v \* A

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-73-2a1ae6ae7fd0> in <module>

----> 1 v \* A

ValueError: operands could not be broadcast together with shapes (3,) (3,4)

* A term called **broadcast** is coming up again and again

**Is it just for multiplication operation?**

**What if we try to add 2 matrices?**

A = np.arange(12).reshape(3, 4) # 12 values, shape 3x4

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

B = np.arange(3).reshape(1, 3) # 3 values, shape 1x3

B

array([[0, 1, 2]])

print(A.shape, B.shape)

(3, 4) (1, 3)

**Adding A and B**

A + B

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-110-151064de832d> in <module>

----> 1 A + B

ValueError: operands could not be broadcast together with shapes (3,4) (1,3)

Again, the term **broadcast** is coming up

**What exactly is broadcast?**

* **Length of row** in **B is 3**
* **Length of each row** in **A is 4**
* It trying to add rows of different lengths, but it is not able to

**Change the shape of A to**3×33×3**and see what happens**

A = np.arange(9).reshape(3, 3) # 9 values, shape 3x3

A

array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

B = np.arange(3).reshape(1, 3) # 2 values, shape 1x3

B

array([[0, 1, 2]])

print(A.shape, B.shape)

(3, 3) (1, 3)

A + B

array([[ 0, 2, 4],

[ 3, 5, 7],

[ 6, 8, 10]])

**Now it is able to add A and B**

* It **added the row of B to each row of A**
* Such operations on np arrays **require a part of the shape to match**
  + Either **no. of rows or no. of columns**
* So that **smaller array can be operated on larger array element-wise**

**This is called “Broadcasting”**

* Broadcasting **allows us to do some operations on 2 arrays of different shapes.**
* The **smaller array** will **repeat itself**, and get **converted to the same shape as of larger array**.

**So, “smaller array gets boradcasted over larger array again and again”**

* Using a different shape for A and B

A = np.arange(12).reshape(4, 3) # 12 values, shape 4x3

A

array([[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8],

[ 9, 10, 11]])

B = np.arange(3).reshape(1, 3) # 2 values, shape 1x3

B

array([[0, 1, 2]])

print(A.shape, B.shape)

(4, 3) (1, 3)

A + B

array([[ 0, 2, 4],

[ 3, 5, 7],

[ 6, 8, 10],

[ 9, 11, 13]])

* **Now B repeats itself 4 times** instead of 3 times when shape of A was (3, 3)

**Observations:**

* **Broadcasting is how numpy treats arrays with different dimensions during arithmetic operations**
* The **smaller array is broadcast across the larger array** so that they have **compatible shapes**.

**We can also use Transpose in some cases**

* To **match the shapes for broadcasting**

A = np.arange(12).reshape(3, 4)

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

B = np.arange(3).reshape(1, 3)

B

array([[0, 1, 2]])

A.shape, B.shape

((3, 4), (1, 3))

**Right now neither no. of rows match nor no. of columns of A and B**

**So, How can we perform addition operation A + B?**

**What if we transpose B?**

* B.T will have 3 rows and 1 column
* Then **no. of rows of A and B.T will be same** —> 3
* So, **B.T can be broadcasted over A** for arithmetic operations

B.T.shape

(3, 1)

A + B.T

array([[ 0, 1, 2, 3],

[ 5, 6, 7, 8],

[10, 11, 12, 13]])

**Few more examples**

**What will be the output of this?**

A = np.array([5, 7, 3, 1])

B = np.array([90, 50, 0, 30])

C = A \* B

A = np.array([5, 7, 3, 1])

A

array([5, 7, 3, 1])

B = np.array([90, 50, 0, 30])

B

array([90, 50, 0, 30])

C = A \* B

C

array([450, 350, 0, 30])

**Why did A \* B work?**

* Array A and B are compatible because of same Dimension

**What will be the output of this?**

A = np.arange(1,9).reshape(3,3)

B = np.array([-1, 0, 1])

A \* B

A = np.arange(1,10).reshape(3,3)

A

array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

B = np.array([-1, 0, 1])

B

array([-1, 0, 1])

A \* B

array([[-1, 0, 3],

[-4, 0, 6],

[-7, 0, 9]])

**Why did A \* B work in this case?**

* A has 3 rows and 3 columns
* B is a 1-D vector with 3 elements
* So, **B gets broadcasted over A for each row of A**

**We can perform other arithmetic operations as well with the help of Broadcasting**

**What will be the result of this?**

A = np.arange(1,10).reshape(3,3)

B = np.arange(3, 10, 3).reshape(3,1)

C = A / B

A = np.arange(1,10).reshape(3,3)

A

array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

B = np.arange(3, 10, 3).reshape(3,1)

B

array([[3],

[6],

[9]])

C = A / B

np.round(C, 1)

array([[0.3, 0.7, 1. ],

[0.7, 0.8, 1. ],

[0.8, 0.9, 1. ]])

**How did this A / B work?**

* A has 3 rows and 3 columns
* B has 3 rows and 1 column —> Its a column vector
* So, **B gets broadcasted on every column of A**

**Generating Random Numbers in Numpy**

**Uniformly Random Distributions**

* **Each number** within a specified range is **equally-likely to be generated**

**We have random module in numpy library**

**randint()**

* For generating random integer value
* It takes **low as starting point**
* **high as ending point** (not included)
* Generates a **random integer between the range (low, high)**

np.random.randint(1, 100)

18

* We can also get an array of randomly generated numbers by **specifying the size**

np.random.randint(1, 100, 5)

array([12, 40, 65, 98, 10])

**rand()**

* Generates a random number,
* **Within default range of (0, 1)**

np.random.rand()

0.2309957471691203

# We can also specify size - number of random numbers we want

np.random.rand(3)

array([0.92702905, 0.91442135, 0.61236261])

**How can we randomly generate a floating point number b/w 50 and 75?**

* We need a **floating point number**
* If we wanted an integer b/w 50 and 75, we would have simply used randint(50, 75)

**So, How can we do this using rand()?**

50 + np.random.rand() \* 25

67.10308157991075

* **rand() gives a floating point number b/w (0, 1)**
* **Size of range** (50, 75) is 75−50=2575−50=25
* **scale the output from rand()**
  + so that it starts generating floating point numbers from (0, 25)

np.random.rand() \* 25

* Now, we need to **shift the range linearly**
  + so that it starts generating floating point numbers from 50, instead of from 0

50 + np.random.rand() \* 25

**Random Normal Distributions**

**In Normal Distribution**

* The probability of generation of numbers follows a **bell curve**
* **mean** and **standard deviation**
* The **mean has highest likelihood of being generated**
* **Values close to mean** have **higher likelihood** of being generated
* **Values farther from mean** value have **lower likelihood** of being generated
* Bell curve is **symmetric around mean value**

**So, enforce generation of random numbers so that they follow a Normal Distribution**

* use np.random.normal()

mu = 100

std = 15

s = np.random.normal(mu, std, 100) # generates 100 values from a Normal Distribution

s

array([ 59.98522692, 122.15750344, 124.46931091, 111.10036561,

100.27053105, 94.50205306, 79.08762572, 94.64240427,

110.90150558, 105.80435665, 121.70800645, 76.37810748,

92.31537171, 100.92186236, 99.60615895, 99.26079765,

92.52042451, 93.92354167, 104.40315076, 106.07721436,

90.72025891, 82.23055024, 88.54189412, 90.71708812,

97.03299987, 81.79948808, 92.61756216, 125.0896788 ,

85.36359884, 89.41282889, 85.33825357, 89.89400632,

107.19451476, 103.97721405, 101.94162478, 85.19948104,

130.37053963, 102.04618471, 101.16226802, 118.31009765,

92.17772962, 99.03407103, 109.49756283, 79.82438503,

104.7385816 , 110.09377519, 90.0199638 , 96.28320306,

87.24665332, 102.14693238, 112.01701927, 81.68930852,

81.49835461, 86.58449169, 98.90163753, 126.31487206,

93.94126854, 109.58492991, 94.64794102, 112.68723699,

114.18952326, 97.50589966, 111.47003532, 112.48260094,

110.85227749, 104.33009771, 97.67089717, 102.91897433,

131.01858145, 103.16803358, 97.95169002, 102.80848965,

107.62357565, 87.79436509, 108.19931545, 78.95842184,

117.06694546, 96.35799526, 102.33281435, 104.73893963,

102.06075459, 109.04087598, 112.37181206, 112.08976161,

86.00674741, 81.26076277, 114.41123769, 92.94062491,

112.2346613 , 64.38216876, 99.15681092, 77.94964367,

114.98154621, 96.05092378, 94.07308966, 104.39452161,

90.24293874, 108.96829528, 88.84312903, 108.60638983])

**If we plot these points against their frequency of generation, they will follow a normal curve**

print(np.mean(s)) # mean of generated points

print(np.std(s)) # std of generated points

99.6743173878269

13.398799497119464

**Shallow vs Deep Copy**

* Numpy **manages memory very efficiently**
* Which makes it really **useful while dealing with large datasets**

**But how does it manage memory so efficiently?**

# We'll create np array

a = np.arange(4)

a

array([0, 1, 2, 3])

# Reshape array `a` and store in b

b = a.reshape(2, 2)

b

array([[0, 1],

[2, 3]])

**Make some changes to our original array a**

a[0] = 100

a

array([100, 1, 2, 3])

b

array([[100, 1],

[ 2, 3]])

* Array **b got automatically updated**

**This is an example of Numpy using “Shallow Copy” of data**

**Now, What happens here?**

* Numpy **re-uses data** as much as possible **instead of duplicating** it
* This helps Numpy to be efficient

**When we created b = a.reshape(2, 2)**

* Numpy **did NOT make a copy of a to store in b**, as we can clearly see
* It is **using the same data as in a**
* It **just looks different (reshaped)** in b
* That is why, **any changes in a automatically gets reflected in b**

**How data is stored using Numpy?**

* Variable **does NOT directly point to data** stored in memory
* There is something called **Header** in-between

**What does Header do?**

* **Variable points to header** and **header points to data** stored in memory
* Header stores **information about data** - called **Metadata**

**a is pointing to Metadata about our data [0, 1, 2, 3], which may include:**

* **How many values** we have --> 4
* What is the **Data Type** of data --> int
* What’s the **Shape** --> (4,)

**When we do b = a.reshape(2, 2)**

* Numpy **does NOT duplicate the data** pointed to by a
* It **uses the same data**
* And **create a New header for b** that **points to the same data** as pointed to by a

**b points to a new Header having different values of Metadata of the same data:**

* **Number of values** --> 4
* **Data Type** --> int
* **Shape** --> (2, 2)

**That is why:**

* When data is accessed using a, it gives data in shape (4,)
* And when data is accessed using b, it gives same data in shape (2, 2)

**This helps Numpy to save time and space - Making it efficient**

**Numpy can also create a “Deep Copy” of data**

a = np.arange(4)

a

array([0, 1, 2, 3])

# Create `c`

c = a + 2

c

array([2, 3, 4, 5])

# We make changes in a

a[0] = 100

a

array([100, 1, 2, 3])

c

array([2, 3, 4, 5])

**As we can see, c did not get affected on changing a**

* Because it is an operation
* A more **permanent change in data**
* So, Numpy **had to create a separate copy for c** - i.e., **deep copy of array a for array c**

**Conclusion:**

* Numpy is able to **use same data** for **simpler operations** like **reshape** —> **Shallow Copy**
* It creates a **copy of data** where operations make **more permanent changes** to data —> **Deep Copy**

**Aggregate / Universal Functions (ufunc)**

* **Ufuncs is short for Universal Functions**
* Ufuncs or universal functions operate on ndarrays in an **element-by-element fashion**.
* They support array **broadcasting**, **type casting**, and several other standard features.
* A ufunc is like a **“vectorized” wrapper** for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

**But, How are these ufuncs different from the functions we saw in last class?**

* “ufunc” is just a term that we gave to **MATHEMATICAL FUNCTIONS in the Numpy library**.
* Numpy provides various universal functions that cover a wide variety of operations.
* These functions **operate on ndarray (N-dimensional array) i.e Numpy’s array class.**
* They perform **fast element-wise array operations**.

**Numpy universal functions are objects that belongs to numpy.ufunc class.**

* Some ufuncs are **called automatically when the corresponding “arithmetic operator” is used on arrays**.
* That’s how they are related to operations on numpy arrays

**For example:**

* When **addition of two array** is performed **element-wise** using + operator, then **np.add() is called internally.**

a = np.array([1,2,3,4])

b = np.array([5,6,7,8])

a+b # ufunc `np.add()` called automatically

array([ 6, 8, 10, 12])

np.add(a,b)

array([ 6, 8, 10, 12])

**There are many unfuncs available in Numpy, like math operations, trigonometric functions, exponents and logarithms, etc.**

**How would calculate the sum of elements of an array?**

**np.sum()**

* It **sums all the values in np array**

a = np.arange(12).reshape(3, 4)

a

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

np.sum(a) # sums all the values present in array

66

**How can we sum the elements in each row or in each column?**

**We can tell np.sum() to add rows or columns spearately**

* By **setting axis parameter**

**What will np.sum(a, axis=0) do?**

* **np.sum(a, axis=0) adds together values in DIFFERENT rows**
* **axis = 0 —> Changes will happen along the vertical axis**
* Summing of values happen **in the vertical direction**
* Rows collapse/merge when we do axis=0

np.sum(a, axis=0)

array([12, 15, 18, 21])

**axis=1**

* **np.sum(a, axis=1) adds together values in DIFFERENT columns**
* **axis = 1 —> Changes will happen along the horizontal axis**
* Summing of values happen **in the horizontal direction**
* Columns collapse/merge when we do axis=1

np.sum(a, axis=1)

array([ 6, 22, 38])

**Now, What if we want to find the average value or median value of all the elements in an array?**

* We have ufuncs like np.mean() and np.median()

**np.mean()**

* np.mean() gives **mean of all values in np array**

np.mean(a)

5.5

**What if we want to find the mean of elements in each row or in each column?**

* We can do **same thing with axis parameter** like we did for np.sum() function

**What will np.mean(a, axis=0) give?**

* It will give **mean of values in DIFFERENT rows**
* **axis = 0 —> Changes will happen along the vertical axis**
* Mean of values will be calculated **in the vertical direction**
* Rows collapse/merge when we do axis=0

np.mean(a, axis=0)

array([4., 5., 6., 7.])

**Now,How can we get mean of elements in each column?**

* **np.mean(a, axis=1) will give mean of values in DIFFERENT columns**
* **axis = 1 —> Changes will happen along the horizontal axis**
* Mean of values will be calculated **in the horizontal direction**
* Columns collapse/merge when we do axis=1

np.mean(a, axis=1)

array([1.5, 5.5, 9.5])

**Now, in a similar way, we can get “median” of all values, or values in different rows, or values in different columns using:**

**np.median()**

* np.median() gives **median of all values in np array**

# same for np.median()

np.median(a, axis=0)

array([4., 5., 6., 7.])

**More useful ufuncs offered by Numpy**

a = np.array([1,2,3,4])

a

array([1, 2, 3, 4])

**Finding if any of the elements in our array is non-zero**

* One way is to do it **manually by iterating through the array**:

def any\_non\_zero(a):

for element in a:

if element != 0:

return True

return False

any\_non\_zero(a)

True

* We defined a function any\_non\_zero()
* It takes an array as input
* It **returns True if any of the elements in array is non-zero**
* It will **return False if none of the elements in array are non-zero, i.e., all elements are zero**

**However, Numpy provides a one-line function to do the same task**

**np.any()**

* any() returns True if **any of the elements** in the argument array is **non-zero**.
* We **don’t have to manually iterate through the array**
* **np.any() does the work for us**

a = np.array([1,2,3,4]) # atleast 1 element is non-zero

np.any(a)

True

a = np.array([1,0,0,0]) # atleast 1 element is non-zero

np.any(a)

True

a = np.zeros(4) # all elements are zero

np.any(a)

False

**np.any() returns False only when there is NO non-zero element present in the array**, i.e., when all array elements are zero

**How to check whether “any” element of array follows a specific condition?**

a = np.array([1,2,3,4])

b = np.array([4,3,2,1])

a, b

(array([1, 2, 3, 4]), array([4, 3, 2, 1]))

**Finding out if any of the elements in array a is smaller than any of the corresponding elements in array b**

**Again, We can find it manually by iterating through both the arrays:**

def any\_smaller(a, b):

for (e1, e2) in zip(a, b): # zip stops the iteration when the shorter of `a` or `b` ends

if e1 < e2:

return True

return False

any\_smaller(a, b)

True

* We defined a function any\_smaller()
* It takes 2 arrays as input
* It **returns True if any of the elements in 1st array is smaller than any of the corresponding elements in 2nd array**
* It will **return False if none of the elements in 1st array are smaller than corresponding elements in 2nd array, i.e., all elements in 1st array are greater than corresponding elements in 2nd array**

**np.any() can become handy here as well**

* Provides a **one line code**
* This is another use case of np.any()
* any() returns True if **any of the corresponding elements** in the argument arrays follow the **provided condition**.

a = np.array([1,2,3,4])

b = np.array([4,3,2,1])

np.any(a<b) # Atleast 1 element in a < corresponding element in b

True

a = np.array([4,5,6,7])

b = np.array([4,3,2,1])

np.any(a<b) # All elements in a >= corresponding elements in b

False

* In this case, **NONE of the elements in a were smaller than their corresponding elements in b**
* So, np.any(a<b) returned False

**So far we saw that np.any() returns True if any, i.e., atleast one element of array is non-zero (default) or follows the specified condition**

**Checking whether “all” the elements in our array are non-zero or follow the specified condition**

* Numpy provides a counterpart of any()
* It’s called all()

**np.all()**

a = np.array([1,2,3,4]) # All elements are non-zero

a

array([1, 2, 3, 4])

**Finding whether or not “all” the elements in our array are non-zero**

* Once again, we can **manually iterate through the array:**

def all\_non\_zero(a):

for element in a:

if element == 0: # if any element is zero, return False

return False

return True # if all elements are non-zero, then only return True

all\_non\_zero(a)

True

b = np.array([1,2,0,4]) # one of the elements is zero

all\_non\_zero(b)

False

* We defined a function all\_non\_zero()
* It takes an array as input
* It **returns False if any of the elements in array is zero**
* It **returns True only if all the elements in array are non-zero**, i.e., NONE of the elements are zero

**This is where Numpy provides the one-line function np.all() to do the same task**

* np.all()
* One use case it all() returns True only if **all elements** in the argument array are **nonzero**.

a = np.array([1,2,3,4]) # All elements are non-zero

a

array([1, 2, 3, 4])

np.all(a)

True

a = np.array([1,2,0,4]) # All elements are NOT non-zero

np.all(a)

False

**Checking whether “all” the elements in our array follow a specific condition**

a = np.array([1,2,3,4])

b = np.array([4,3,2,1])

a, b

(array([1, 2, 3, 4]), array([4, 3, 2, 1]))

**Finding out if all the elements in array a are smaller than all the corresponding elements in array b**

**We can find it manually by iterating through both the arrays:**

def all\_smaller(a, b):

for (e1, e2) in zip(a, b): # zip stops the iteration when the shorter of `a` or `b` ends

if e1 > e2:

return False

return True

all\_smaller(a, b)

False

a = np.array([1,0,0,0])

b = np.array([4,3,2,1])

all\_smaller(a, b)

True

* We defined a function all\_smaller()
* It takes 2 arrays as input
* It **returns False if any of the elements in 1st array is greater than any of the corresponding elements in 2nd array**
* It will **return True only if all the elements in 1st array are smaller than corresponding elements in 2nd array**

**This is where np.all() comes handy**

* It provides a one-line code to check for the condition
* all() **returns True only if all of the corresponding elements in the argument arrays** follow the **provided condition**.

a = np.array([1,2,3,4])

b = np.array([4,3,2,1])

a, b

(array([1, 2, 3, 4]), array([4, 3, 2, 1]))

np.all(a<b) # Not all elements in a < corresponding elements in b

False

a = np.array([1,0,0,0])

b = np.array([4,3,2,1])

np.all(a<b) # All elements in a < corresponding elements in b

True

* In this case, **ALL the elements in a were smaller than their corresponding elements in b**
* So, np.all(a<b) returned True

**Numpy is a very vast library**

* There are a lot more ufuncs than the ones we saw today
* You can explore other methods for performing different mathematical tasks on your own
* **We’ll also provide some more useful ufuncs for you to read and explore after this lecture**
* We will use numpy in almost every lecture of DS and ML from now on

**3 Dimensional Arrays**

B = np.arange(24).reshape(2, 3, 4)

B

array([[[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]],

[[12, 13, 14, 15],

[16, 17, 18, 19],

[20, 21, 22, 23]]])

**How many dimensions B has?**

* 3
* It’s a **3-dimensional tensor**

**How is reshape(2, 3, 4) working?**

* If you see, it is giving 2 matrices
* Each matrix has 3 rows and 4 columns

**So, that’s how reshape() is interpreted for 3D**

* **1st argument** gives **depth** (No. of Matrices)
* **2nd agrument** gives **no. of rows** in each depth
* **3rd agrument** gives **no. of columns** in each depth

**Getting just the whole of 1st Matrix**

B[0]

**What value will we get if we do B[0, 0, 0]?**

B[0, 0, 0]

#### What value will we get if we do `B[1, 1, 1]`?

B[1, 1, 1]

# It looks at Matrix 1, that is, 2nd Matrix (Not Matrix 0)

# Then it looks at row 1 of matrix 1

# Then it looks at column 1 of row 1 of matrix 1

**We can also Slicing in 3-Dimensions**

* Works same as in 2-D matrices

**Use Case: Image Manipulation using Numpy**

* By now, you already have an idea that Numpy is an amazing open-source Python library for **data manipulation** and **scientific computing**.
* It is used in the domain of **linear algebra**, Fourier transforms, **matrices**, and the **data science field**.
* **NumPy arrays are way faster than Python Lists**.

**Using Numpy for Image Processing**

* The fundamental idea is that we know **images are made up of Numpy ndarrays**.
* So we can **manipulate these arrays and play with images**.
* This use case is to give you a broad overview of **Numpy for Image Processing.**

import numpy as np

import matplotlib.pyplot as plt

**Playing with images using Numpy**

**Opening an Image**

* We first need to open it
* To open an image, we will use the matplotlib library to read and show images.
* For this use case, just know that it uses an image module for working with images.
* It offers two useful methods **imread() and imshow()**.

**imread() – to read the images**

**imshow() – to display the images**

**Drive link for the image:**

Download the image fruits.jpg from here: <https://drive.google.com/file/d/1lHPQUi3wdB6HxN-SNJSBQXK7Z0y0wf32/view?usp=sharing>

and place it in your current working directory

img = np.array(plt.imread('fruits.png'))

plt.imshow(img)

<matplotlib.image.AxesImage at 0x7f9942583a60>



**Details of an Image**

**Dimensions and shape of this image**

print('# of dims: ',img.ndim) # dimension of an image

print('Img shape: ',img.shape) # shape of an image

# of dims: 3

Img shape: (1333, 2000, 3)

**2-D image has 3 dimensions?**

* **Coloured images have a 3rd dimension for depth or RGB colour channel**
* Here, the **depth is 3**
* When we discussed **3-D Arrays**, we saw that **depth was the first element of the shape tuple**
* But when we are loading an image using **matplotlib and getting its 3-D array**, we see that **depth is the last element of the shape tuple**

**Why is there a difference b/w normal np array and the np array generated from Matplotlib in terms of where the depth part of shape appears?**

* This is how matplotlib reads the image
* It **reads the depth values (R, G and B values) of each pixel one by one** and stacks them one after the other

**The shape of imge we read is: (1333, 2000, 3)**

* matplotlib **first reads that each plane has**1333×20001333×2000**pixels**
* Then, it **reads depth values (R, G and B values) of each pixel and place the values in 3 separate planes**
* That is why **depth is the last element of shape tuple in np array generated from an image read by matplotlib**
* Whereas in a **normal np array, depth is the first element of shape tuple**

**Visualizing RGB Channels**

We can split the image into each RGB color channels using only Numpy

**But, What exactly RGB values are?**

* These are values of each pixel of an image
* Each pixel is made up of **3 components/channels** - **Red, Green, Blue** - which form RGB values
* Coloured images are usually stored as 3-dimensional arrays of **8-bit unsigned integers**
* So, the range of values that each channel of a pixel can take is 00 to 28−128−1
* That is, each pixel’s each channel, R, G and B can range from **0 to 255**

**Each pixel has these 3 values which combined together forms the colour that the pixel represents**

* So, a pixel **[255, 0, 0 ]** will be **RED** in colour
* A pixel **[0, 255, 0]** will be **GREEN** in colour
* A pixel **[0, 0, 255]** will be **BLUE** in colour

**What will be the colour of pixel [0, 0, 0]?**

* Black

**What will be the colour of pixel [255, 255, 255]?**

* White

**Separating the R, G, B channels in our image:**

* We’ll make use of **slicing of arrays**
* For **RED** channel, we’ll **set values of GREEN and BLUE to 0**
* Similarly, for GREEN channel, we’ll set values of RED and BLUE to 0
* … and same for BLUE channel

img = np.array(plt.imread('fruits.png'))

img\_R, img\_G, img\_B = img.copy(), img.copy(), img.copy()

img\_R[:, :, (1, 2)] = 0

img\_G[:, :, (0, 2)] = 0

img\_B[:, :, (0, 1)] = 0

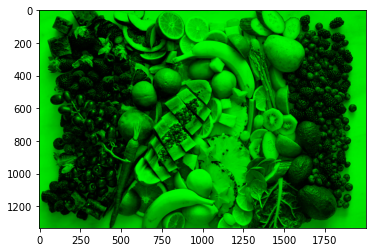
plt.imshow(img\_R)

<matplotlib.image.AxesImage at 0x7f99133fd0d0>



plt.imshow(img\_G)

<matplotlib.image.AxesImage at 0x7f9937a31130>



plt.imshow(img\_B)

<matplotlib.image.AxesImage at 0x7f993a45f6a0>



**Converting Image to Gray Scale**

* We can also convert a coloured image to gray scale using Numpy

**Gray Scale image**

* A gray scale image is one which **only has 1 channel** - Either R or G or B
* We can get images having a single channel using the concept of **slicing in Numpy**

**How will we slice the image array to get images having single channel**

* We need all rows and columns, but we only need 1 channel at a time
* **RED channel** is at **depth index 0**
* **GREEN channel** is at **depth index 1**
* **BLUE channel** is at **depth index 2**

**For example, to get grayscale image only for RED channel:**

image[:, :, 0]

* image[:, :, 0] means get **all rows, all columns**, and the **first (at index 0) color channel**, which is the **RED channel**.

img = np.array(plt.imread('fruits.png'))

img\_R\_grayscale = img[:,:,0]

plt.imshow(img\_R\_grayscale, cmap='gray')

# Matplotlib uses a colormap by default when plotting a single channel.

# So, we have to specify `cmap='gray' within the `imshow() function

<matplotlib.image.AxesImage at 0x7f991263bd90>



* As you can see the **areas with RED colour in original image are more bright in gray scale image** with only R channel
* We can use this information to indentify where a **colour channel is more concentrated within the image**
* Because the area with that colour channel will **brighten up in gray scale image**

img\_R\_grayscale.shape

(1333, 2000)

**As you can notice, its a 2-D array now**

* Because **depth of R, G, B has been removed**
* It **only consists of 1 channel** - R channel

**GREEN channel is at depth index 1 and BLUE channel is at depth index 2**

* **image[:, :, 1]** —> **Gray Scale image** with **only GREEN** channel
* **image[:, :, 2]** --xx-> **Gray Scale image** with **only BLUE** channel

img\_G\_grayscale = img[:,:,1]

plt.imshow(img\_G\_grayscale, cmap='gray')

<matplotlib.image.AxesImage at 0x7f991331a670>



**Do you see the difference b/w img\_R\_grayscale and img\_G\_grayscale?**

* Brightness of different areas is different
* In img\_R\_grayscale, the red areas are more bright
* In img\_G\_grayscale, the green areas are more bright

**Gray scale image with only BLUE channel:**

img\_B\_grayscale = img[:,:,2]

plt.imshow(img\_B\_grayscale, cmap='gray')

<matplotlib.image.AxesImage at 0x7f99161a1970>



Now, you can clearly see the differences b/w gray scale images having each channel separately

**Negative of an Image**

* Converting a color image to a negative image is very simple.
* We just have to **change/invert the RGB values of each pixel** in image.
* We perform only **3 steps for each pixel** of the image:
  1. First, **get the RGB values** of the pixel
  2. **Calculate new RGB values** using **R = 255 – R, G = 255 – G, B = 255- B**
  3. Finally, **save the new RGB values** in the pixel

**We’ll use a new image to better display how we can get the negative of an original image**

Download the image emma\_stone.jpg from here: <https://drive.google.com/file/d/12H8yrRImTt43b-EV22BQsaRiYjbadyI-/view?usp=sharing>

and place it in your current working directory

img = np.array(plt.imread('emma\_stone.jpg'))

plt.imshow(img)

<matplotlib.image.AxesImage at 0x7f99230255b0>



* Things like arr + 2, arr - 4, arr \* 2 —> would **perform operation on each element**
* We can use the same concept to get new pixel values by doing **255 - arr**
* This would **subtract each R, G and B value from 255** and **give negative of the image**

img\_neg = 255 - img

plt.imshow(img\_neg)

<matplotlib.image.AxesImage at 0x7f9920e73b80>



**Rotating an Image (Transpose the Numpy Array)**

* **Rotating the image means transposing the array**

**For this, we’ll use the np.transpose() function in numpy**

* It takes 2 arguments

**1st argument** is obviously the **array that we want to transpose (image array in our case)**

**2nd argument is axes**

* Its a **tuple or list of ints**
* It contains a **permutation of [0,1,…,N-1] where N is the number of axes of array**

**Now, our image array has 3 axes (3 dimensions) —> 0th, 1st and 2nd**

* We specify how we want to transpose the array by giving an **order of these axes inside the tuple**
  + **Vertical axis (Row axis) is 0th axis**
  + **Horizontal axis (Column axis) is 1st axis**
  + **Depth axis is 2nd axis**
* **In order to rotate the image, we want to transpose the array**
* That is, we want to **transpose rows into columns and columns into rows**
* So, we want to **interchange the order of row and column axis** —> **interchange order of 0th and 1st axis**
* We **don’t want to change the depth axis (2nd axis)** —> So, it will **remain at its original order position**

Now, the **order of axes in orginal image is (0, 1, 2)**

**What will be the order of axes rotated image or transposed array?**

* The **order of axes in rotated image will be (1, 0, 2)**
* **Order (Position) of 0th and 1st column is interchanged**

img = np.array(plt.imread('emma\_stone.jpg'))

img\_rotated = np.transpose(img, (1,0,2))

plt.imshow(img\_rotated)

<matplotlib.image.AxesImage at 0x7f9920d186d0>



* We obtained the **rotated image by transposing the np array**

**Trim Image**

* So, We can trim/crop an image in Numpy using Array using **Slicing**.

img = np.array(plt.imread('emma\_stone.jpg'))

plt.imshow(img)

<matplotlib.image.AxesImage at 0x7f9923074190>



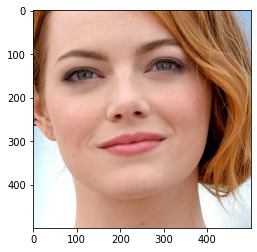
* If you see x and y axis, the face starts somewhat from ~500 and ends at ~1000 on x-axis
  + **x-axis in image is column axis in np array**
  + Columns change along x-axis
* And it lies between ~200 to ~700 on y-axis
  + **y-axis in image is row axis in np array**
  + Rows change along y-axis

**We’ll use this information to slice our image array**

img\_crop = img[200:700, 500:1000, :]

plt.imshow(img\_crop)

<matplotlib.image.AxesImage at 0x7f99233c1fd0>



**Saving Image as ndarray**

To save a ndarray as an image, we can use matplotlib’s plt.imsave() method.

* **1st agrument** —> We provide the path and name of file we want to save the image as
* **2nd agrument** —> We provide the image we want to save

path = 'emma\_face.jpg'

plt.imsave(path, img\_crop)

**Now, if you go and check your current working directory, image would have been saved by the name emma\_face.jpg**

This way we can do many interesting things with Images using just Numpy, without the use of image processing libraries like OpenCV.

Last changed by

**Scaler Topics**Scaler Topics

 289 [1](https://hackmd.io/@scaler-topics-main/rkNhoi6gq)

Published on [**HackMD**](https://hackmd.io/)

**Supplementary Post Read for Numpy**

In this reading, we’ll cover some more useful functionality provided by Numpy

**Content**

* **Absolute values**
  + np.absolute()
  + np.abs()
* **Partitioning**
  + np.partition()
  + np.argpartition()
* **Some more useful ufuncs**
* **Trigonometric Functions**
  + np.sin(), np.cos()
* **Exponential and Logarithmic Functions**
  + np.exp(), np.log(), np.log2(), np.log10()

import numpy as np

**Arrays Filled with Sequences**

**np.linspace()**

In numerical computing it is very common to require arrays with evenly spaced values between a starting value and ending value. Numpy provides two similar functions to create such arrays: np.arange() and np.linspace(). Both functions take three arguments, where the first two arguments are the start and end values.

The third argument of np.arange() is the increment, while for **np.linspace() it is the total number of points in the array**.

We already saw np.arange() in class.

**Let’s see np.linspace() now:**

np.linspace(0, 10, 11)

array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])

It created an array having 11 elements where start value is 0 and end value is 10.

Now, whether to use np.arange or np.linspace is mostly a matter of personal preference, but it is generally recommended to use np.linspace() whenever the increment is a non-integer.

**np.logspace()**

The function np.logspace() is similar to np.linspace(), but the **increments between the elements in the array are logarithmically distributed**, and the first two arguments, for the **start** and **end** values, are the**powers of the optional base keyword argument (which defaults to 10)**.

**For example, to generate an array with logarithmically distributed values between 1 and 100, we can use:**

np.logspace(0, 2, 5) # 5 data points between 10\*\*0=1 to 10\*\*2=100

array([ 1. , 3.16227766, 10. , 31.6227766 ,

100. ])

**Absolute values**

At times, we might need to find absolute values of elements in array. Numpy provides a very easy-to-use function for this purpose

**np.absolute()**

It calculate the absolute value element-wise. It returns an ndarray containing the absolute value of each element.

x = np.array([-1.2, 1.2])

np.absolute(x)

array([1.2, 1.2])

If the input is a complex value, like x = a + ib, the absolute value is √(a2+b2)(a2+b2). This is a scalar if x is a scalar.

np.absolute(1.2 + 1j)

1.5620499351813308

**np.abs()**

The abs function can be used as a shorthand for np.absolute on ndarrays.

x = np.array([-1.2, 1.2])

np.abs(x)

array([1.2, 1.2])

**Partitioning**

We can partially sort Numpy arrays using the functions provided in the library. Let’s see how:

**np.partition()**

This method returns a partitioned copy of an array.

It creates a copy of the array with its elements rearranged in such a way that the value of the element in k-th position is in the position it would be in a sorted array. All elements smaller than the k-th element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

**Let’s see it in action:**

a = np.array([3, 4, 2, 1])

a

array([3, 4, 2, 1])

np.partition(a, 3)

array([2, 1, 3, 4])

3rd element in the sorted array would have been 3. So 3 is placed at the 3rd position (index 2). Rest all the elements are placed in such a manner that all elements less than partitioning element are placed before it and all elements greater than or equal to partitioning element are placed after it. The ordering of elements in the two partitions is not fixed. So, it’s not sorting.

Now, if we provide with a sequence of k-th instead of a single integer value like above, it will partition all elements indexed by k-th of them into their sorted position at once. Let’s see it:

a = np.array([32, 50, 27, 10, 43])

np.partition(a, (1, 3))

array([10, 27, 32, 43, 50])

1st element (index 0) in the sorted array would have been 10 and 3rd element (index 2) in the sorted array would have been 32. So, 10 and 32 are placed in the positions where they would have been in a sorted array. Rest all elements are placed such that all elements less than 10 are before it, all elements greater than 10 are after it, all elements less than 32 are before it and all elements greater than 32 are after it.

**np.argpartition()**

It works just like agrsort() we saw in the lecture. It perform an indirect partition along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as the array that index data along the given axis in partitioned order.

**Let’s see its working:**

x = np.array([30, 40, 20, 10])

np.argpartition(x, 3)

array([2, 3, 0, 1])

This is an indirect partitioning using indices. Instead of array of elements, it gives a paritoned array of orginial indices of those elements.

The element 30 at index 0 in original array would have been at 3rd position in sorted array. So, 0 is placed at 3rd position. Rest all original indices are arranged such that indices whose corresponding elements are less than the partitioning element are before that index 0 and indices whose corresponding elements are greater than the partitioning element are after that index 0.

**Some more useful ufuncs for you**

**Trigonometric Functions**

In addition to arithmetic expressions using operators, Numpy provides functions for element-wise evaluation of many elementary trigonometric functions and operations.

Each of these functions takes a single array (of arbitrary dimension) as input and returns a new array of the same shape, where for each element the function has been applied to the corresponding element in the input array.

**np.sin(), np.cos()**

This function takes only one argument and is used to compute the sine function for all values in the array:

x = np.linspace(-1, 1, 11)

x

array([-1. , -0.8, -0.6, -0.4, -0.2, 0. , 0.2, 0.4, 0.6, 0.8, 1. ])

y = np.sin(np.pi \* x)

np.round(y, decimals=4)

array([-0. , -0.5878, -0.9511, -0.9511, -0.5878, 0. , 0.5878,

0.9511, 0.9511, 0.5878, 0. ])

Here we also used the constant np.pi and the function np.round() to round the values of y to four decimals.

Like the np.sin function, many of the elementary trigonometric math functions take one input array and produce one output array. We can also make these functions operate on two input arrays and return one array:

**For example:**sin2x+cos2x=1sin2⁡x+cos2⁡x=1

np.add(np.sin(x) \*\* 2, np.cos(x) \*\* 2)

array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

np.sin(x) \*\* 2 + np.cos(x) \*\* 2

array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

**Exponential and Logarithmic Functions**

**np.exp()**

This function returns element-wise exponent raised to power of element’s value

x = np.arange(0,3)

x

array([0, 1, 2])

np.exp(x) # returns e\*\*0, e\*\*1, e\*\*2

array([1. , 2.71828183, 7.3890561 ])

**np.log(), np.log2(), np.log10()**

These functions return Logarithms of base e, 2, and 10, respectively.

x = np.arange(1,11)

x

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

np.log(x)

array([0. , 0.69314718, 1.09861229, 1.38629436, 1.60943791,

1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509])

np.log10(x)

array([0. , 0.30103 , 0.47712125, 0.60205999, 0.69897 ,

0.77815125, 0.84509804, 0.90308999, 0.95424251, 1. ])

Numpy is really a vast library. There are a lot of functions provided by it, all of which may not be covered in the lecture or in this reading.

However, we’ll introduce and explain the functions as and when they will be used in future. Meanwhile, feel free to explore Numpy on your own and carry out some interesting computations.

Last changed by

**Scaler Topics**Scaler Topics

 238

Published on [**HackMD**](https://hackmd.io/)